ELSEVIER

Theoretical Computer Science 164 (1996) 1–12

Theoretical Computer Science

Heaps with bits

Svante Carlsson [a], Jingsen Chen [a,*], Christer Mattsson [b]

[a] *Department of Computer Science, Luleå University, S-971 87 Luleå Sweden*
[b] *Quality Laboratories AB, IDEON Research Park, S-223 70 Lund, Sweden*

Received March 1995; revised July 1995
Communicated by M. Nivat

Abstract

In this paper, we show how to improve the complexity of heap operations and heapsort using extra bits. We first study the parallel complexity of implementing priority queue operations on a heap. The trade-off between the number of extra bits used, the number of processors available, and the parallel time complexity is derived. While inserting a new element into a heap in parallel can be done as fast as parallel searching in a sorted list, we show how to delete the smallest element from a heap in constant time with a sublinear number of processors, and in sublogarithmic time with a sublogarithmic number of processors. The models of parallel computation used are the CREW PRAM and the CRCW PRAM. Our results improve those of previously known algorithms. Moreover, we study a variant, the fine-heap, of the traditional heap structure. A fast algorithm for constructing this new data structure is designed using an interesting technique, which is also used to develop an improved heapsort algorithm. Our variation of heapsort is faster than Wegener's heapsort and requires less extra space.

1. Introduction

One of the fundamental data types in Computer Science is the priority queue. It has been useful in many applications [11]. A *priority queue* is a set of elements on which two basic operations are defined: inserting a new element into the set and deleting the minimum element from the set. Several data structures have been proposed for implementing priority queues. Probably the most elegant one is the heap [21]. A (min-)heap is a binary tree with *heap-property*: (*i*) It has the *heap shape*; i.e., all leaves lie on at most two levels which are adjacent and all leaves on the last level occupy the leftmost positions and all other levels are complete; (*ii*) It is *min-ordered*: the key value associated with each node is not smaller than that of its parent. The minimum element is then at the root, which is at the first level. We refer to the number of elements in a heap as its *size*. A max-heap is defined similarly.

* Corresponding author. E-mail: svante,jingsen@sm.luth.se.

SSDI 0304-3975(95)00152-2

The problem of heap construction and heap operations have received considerable attention in the literature [2, 5, 6, 8, 9, 11, 13]. In the parallel models of computation, optimal heap construction algorithms have also been developed [4, 15]. However, parallel heap operations have not been so deeply studied. Recently, Pinotti and Pucci [14] presented an $\mathcal{O}(\log\log n)$-time [1] parallel algorithm for deleting the smallest element from a heap of size n using $n/\log n$ EREW-PRAM processors; and Zhang and Korf [22] reduced the number of processors used for the deletion to $(n/\log n)^{1-1/k}$ for some constant k, $1 \leqslant k \leqslant \lceil \log(n/\log n) \rceil$. In this paper, the trade-off between the number of extra bits used, the number of processors available, and the parallel-time complexity of heap operations is investigated. We first present a constant-time parallel deletion algorithm on the concurrent-read concurrent-write (CRCW) PRAM model. On this model, a multiple-write access to the same memory location succeeds only when all the processors writing to that cell are attempting to write the same value. Next, we show how to perform a delete operation in a heap of size n in $\mathcal{O}(\log n/\log\log n)$ time using $\log n/\log\log n$ processors on the same model. All our CRCW-PRAM algorithms use n extra bits. Moreover, if $n\log n$ extra bits are available and if a processor can write 0 or 1 into the bit of a word (where a word is of $\lfloor \log n \rfloor$ bits), the complexity of our parallel algorithms remains the same on the concurrent-read exclusive-write (CREW) PRAM model.

Sorting is a fundamental algorithmic problem. One of the well studied in-place sorting algorithm is the heapsort, which first constructs a heap on the input elements and then deletes the elements one by one from the heap. The classical heapsort [8, 21] needs $2n\log n + \mathcal{O}(n)$ comparisons both in the worst and average cases to sort n elements [16]. During the past 30 years, several variants of heapsort have been developed [2, 3, 9, 10, 13, 19, 20]. The fastest one is the variant proposed by Wegener [19], which takes $n\log n + 1.1n$ (and $n\log n + n$ for $n = 2^h - 1$) comparisons in the worst case, using n bits of extra storage. Moreover, it also makes $\mathcal{O}(n\log n)$ two-bit variable comparisons. This is very close to the information-theoretic lower bound of $n\log n - 1.4427n$ comparisons for sorting n elements. In Section 3, we shall study a new variant, the fine-heap, of the traditional heap structure, which is a heap with additional ordering relation defined on siblings. Efficient construction algorithm for this new structure is presented. This algorithm is not only simple and fast, but also employs a powerful technique for designing comparison-based algorithms (namely, mass productions, which was previously used for designing the fastest known selection algorithm [17]). With the fine-heap, we show how to obtain a variant of Wegener's heapsort and achieve an upper bound of $n\log n + 1.00274n$ (and $n\log n + 0.91667n$ for $n = 2^h - 1$) comparisons in the worst case. Furthermore, our variant requires either $\lfloor n/2 \rfloor$ extra bits and $\mathcal{O}(n\log n)$ two-bit variable comparisons, or n extra bits and no bit comparisons. Remark that parallel algorithms for heap construction and heap operations can be adapted to the fine-heap, which may also result in a parallel version of heapsort.

[1] All logarithms in this paper are to base 2.

2. Parallel heap operations

Notice that a heap on n elements can be stored level by level from left to right in an array $\mathscr{H}$ with the property that the element at position i has its parent at $\lfloor i/2 \rfloor$ and its children at $2i$ and $2i+1$. Thus, the addresses of all the nodes on a path from the root to some leaf of a heap can easily be computed by shift operations. The level, $level(\mathscr{H}[i])$, of an element $\mathscr{H}[i]$ in the heap $\mathscr{H}$ is defined as $\lfloor \log i \rfloor + 1$.

For the insertion of a new element x into a heap $\mathscr{H}$ of size n, an optimal sequential algorithm of $\mathcal{O}(\log \log n)$ comparisons works as follow: First, x is placed at the first available position $\mathscr{H}[n+1]$; and then the min-ordering is (re)stored on the path from $\mathscr{H}[1]$ down to $\mathscr{H}[n+1]$. This is equivalent to the problem of searching x in the path from $\mathscr{H}[1]$ to $\mathscr{H}[\lfloor (n+1)/2 \rfloor]$ (which form a sorted list). For the complexity of parallel searching, see [12, 18]. Therefore, the following observation is immediate.

Observation 2.1. *The parallel complexity of the insert operation in a heap of size n is the same as that of searching in a sorted list of length* $\lfloor \log(n+1) \rfloor$ *on the same model of parallel computation.*

The delete operation in a heap $\mathscr{H}[1..n]$ consists of first removing the smallest element from the heap, replacing it with $\mathscr{H}[n]$, and then restoring the min-ordering property. The sequential deletion can be done optimally in logarithmic time. However, it appears that the delete operation is inherently sequential, since the operation involves the search of $\mathscr{H}[n]$ in some path from either $\mathscr{H}[2]$ or $\mathscr{H}[3]$ down to the leaf-level (called the path of minimum children). Hence, the deletion may not admit an efficient parallel solution. Observing that the searching path for $\mathscr{H}[n]$ is not known beforehand, we have

Observation 2.2. *In a heap, the parallel complexity of the delete operation is at least as hard as that of parallel insertion.*

In the rest of this section, nevertheless, we will try to parallelize the delete operation. More precisely, we shall demonstrate that it is possible to perform the deletion in constant time with a sublinear number of processors, and in sublogarithmic time using a sublogarithmic number of processors. Assume without loss of generality that the size of the heap is of form $2^h - 1$ for some integer $h \geqslant 0$. Otherwise, one can first perform the parallel deletion on the first $\lfloor \log n \rfloor$ levels of the heap and then in $\mathcal{O}(1)$ steps find out which element at the leaf-level of the heap is the last node of the path of minimum children. We first show that the root deletion in a heap of size n can be solved in constant time with $\mathcal{O}(n \log n)$ processors on the CRCW PRAM model.

Algorithm 2.3. *Suppose the number of processors available is*

$$p = (n+1)/2(\lfloor \log n \rfloor - 1).$$

1. *Associate with the heap* $\mathscr{H}$ *an array of n bits, denoted by* $\mathscr{B}[1..n]$. *Initially, let* $\mathscr{B}[i] = 0$ *for* $i = 1, 2, \ldots, n$;
2. *For each* i, $1 \leqslant i \leqslant (n-3)/4$, *a processor* $\mathscr{P}_i$ *is assigned to the element* $\mathscr{H}[2i]$ *of the heap. The processor* $\mathscr{P}_i$ *reads* $\mathscr{H}[2i]$ *and* $\mathscr{H}[2i+1]$, *determines the smaller one, and stores the value* 1 *either in* $\mathscr{B}[2i]$ *if* $\mathscr{H}[2i] < \mathscr{H}[2i+1]$ *or in* $\mathscr{B}[2i+1]$ *otherwise*;
3. *For each* i, $2 \leqslant i \leqslant (n-1)/2$, k_i ($\triangleq (n+1)/2^{\lfloor \log i \rfloor + 1}$) *processor(s) are assigned to the element* $\mathscr{H}[i]$ *of the heap*;
4. *For every leaf node* $\mathscr{H}[j]$ ($(n+1)/2 \leqslant j \leqslant n$), *let* $\mathscr{P}_i^{(j)}$ *be the processor that is assigned to its parent* $\mathscr{H}[\lfloor j/2^i \rfloor]$ ($1 \leqslant i \leqslant \lfloor \log n \rfloor - 1$) *according to* $\mathscr{H}[j]$ *in the preceding step.* $\mathscr{P}_i^{(j)}$ *reads the value of* $\mathscr{B}[\lfloor j/2^i \rfloor]$ ($1 \leqslant i \leqslant \lfloor \log n \rfloor - 1$) *and writes it in* $\mathscr{B}[j]$. *All the read and write accesses are done simultaneously for all the processors*;
5. *Compute the path where* $\mathscr{H}[n]$ *will be located, which is from the root of the heap to a leaf either* $\mathscr{H}[j]$ *or* $\mathscr{H}[j+1]$ *according as* $\mathscr{H}[j] < \mathscr{H}[j+1]$ *or not, where* $\mathscr{H}[j]$ *and* $\mathscr{H}[j+1]$ *are the leaves of the heap with* $\mathscr{B}[j] = \mathscr{B}[j+1] = 1$;
6. *Search for* $\mathscr{H}[n]$ *on the path founded in the above step.*

The correctness of Algorithm 2.3 follows immediately from the fact that our parallel deletion is obtained by implementing the sequential deletion algorithm on the PRAM model. Moreover, the contents of $\mathscr{B}$ can be updated easily and fast. Clearly, only Step 4 of Algorithm 2.3 requires $\mathcal{O}(n \log n)$ processors. After Step 4 is completed, at most two leaf-elements whose associated bits are 1 and these two leaves are brothers. Now, by using $\mathcal{O}(\log n)$ processors:

- The path of minimum children can be computed in $\mathcal{O}(1)$ time (i.e., Step 5);
- The insertion of $\mathscr{H}[n]$ on the path of minimum children takes $\mathcal{O}(1)$ time (i.e., Step 6), employing a constant-time parallel searching algorithm [1].

Since Steps 1–4 of Algorithm 2.3 can also be done in constant time, we have

Lemma 2.4. *There is a CRCW-PRAM algorithm for deleting the smallest element in a heap of size n, running in* $\mathcal{O}(1)$ *time with* $\mathcal{O}(n \log n)$ *processors and* $\mathcal{O}(n)$ *extra bits.*

Remark that Algorithm 2.3 only demands the the ability of multiple-write for its fourth step, which can be modified for running on the CREW-PRAM model. Assume that $\mathcal{O}(n \log n)$ extra bits are available and that a processor can write a digit into the bit of a word. Now, we associate with each leaf node $\mathscr{H}[j]$ ($(n+1)/2 \leqslant j \leqslant n$) of the heap $\mathscr{H}$ a word W_j of $\lfloor \log n \rfloor$ bits; denote by $W_j(i)$ the ith bit of W_j. Initially, let $W_j(i) = 0$ for all $(n+1)/2 \leqslant j \leqslant n$ and $1 \leqslant i \leqslant \lfloor \log n \rfloor$. Each processor $\mathscr{P}_i$ assigned to $\mathscr{H}[i]$ ($2 \leqslant i \leqslant n$) compares $\mathscr{H}[i]$ with its sibling and writes the value 1 in the ($\lfloor \log n \rfloor$)th bit of all the words W_j, where W_j is the word associated to $\mathscr{H}[j]$ that is the leaf node in the subheap rooted at $\mathscr{H}[i]$. Then, the leaf node $\mathscr{H}[k]$ whose associated word W_k has the value 1 in all its bits is the last element on the path of minimum children.

After that, the path of minimum children can easily be computed in constant time. We thus have

Lemma 2.5. *There is a CREW-PRAM algorithm for deleting the smallest element in a heap of size n, running in* $\mathcal{O}(1)$ *time with* $\mathcal{O}(n \log n)$ *processors and* $\mathcal{O}(n \log n)$ *extra bits.*

Notice that the sequential complexity of the deletion is $\mathcal{O}(\log n)$. Hence, the above schemes fail to achieve the optimal speedup. However, the number of processors required can be reduced significantly. First, some definitions are needed. We denote by $\|S\|$ the cardinality of a set S of elements. An integer $m > 0$ is called a *perfect number* if there exists some integer $k > 0$ such that $m = 2^k - 1$. For any integer $m > 0$, we define the *left match* m' of m as the largest perfect number such that $m' \leqslant m$. Notice that if m itself is a perfect number, then $m' = m$.

Algorithm 2.6. *Suppose that the number of processors available is* $p = n$ *and that* $\mathcal{H}$ *is a heap of size n. Let k be the left match of* $\lfloor n/\log n \rfloor$.

1. *perform a parallel deletion on the first* $\log k$ *levels of* $\mathcal{H}$ *(with* $\mathcal{H}[n]$ *being the element to be inserted), called the heap* $\mathcal{H}_u$, *using p processors. Call the heap* $\mathcal{H}_u$ *after the deletion by* $\mathcal{H}'_u$;
2. *compute the path of minimum children in the heap* $\mathcal{H}'_u$ *and denote the last element on this path by z. Let z' be the smaller child of z in the heap* $\mathcal{H}$;
3. *run recursively a parallel deletion algorithm on the subheap,* $\mathcal{H}_D$, *rooted at z' in the heap* $\mathcal{H}$ *using p processors.*

The algorithm above deletes the smallest element in a heap correctly, which is similar to the sequential deletion algorithm. Moreover, the running time of Algorithm 2.6 is $\mathcal{O}(1)$. Notice first that

$$\|\mathcal{H}_u\| \cdot \log \|\mathcal{H}_u\| \leqslant \frac{n}{\log n} \cdot \log\left(\frac{n}{\log n}\right) \leqslant n$$

and $$\|\mathcal{H}'_u\| \cdot \log \|\mathcal{H}'_u\| \leqslant \frac{n}{\log n} \cdot \log\left(\frac{n}{\log n}\right) \leqslant n.$$

Therefore, Steps 1 and 2 of the algorithm run in constant time by Lemma 2.4. Notice next that the deletion on $\mathcal{H}_D$ can be done in $\mathcal{O}(1)$ time since $\|\mathcal{H}_D\| = \mathcal{O}(\log n)$. (Remark that if Steps 1 and 2 of Algorithm 2.6 are performed according to Lemma 2.5, the number of extra bits needed is then $\mathcal{O}(n)$.) Therefore,

Lemma 2.7. *There is a parallel algorithm for deleting the smallest element in a heap of size n, running in* $\mathcal{O}(1)$ *time with* $\mathcal{O}(n)$ *CRCW- (or CREW-) PRAM processors and* $\mathcal{O}(n/\log n)$ *(or* $\mathcal{O}(n)$ *for the CREW-PRAM model) extra bits.*

To further reduce the number of processors consumed, we shall employ $p = \lceil n^\varepsilon \rceil$ processors, where $0 < \varepsilon < 1$ is a constant, and let k be the left match of $\lceil n^\varepsilon \rceil$. By executing three steps of Algorithm 2.6 with the new values of p and k, Steps 1 and 2 are completed in $\mathcal{O}(1)$ time as well. For Step 3, notice that the algorithm will now be repeated $\mathcal{O}(1/\varepsilon)$ times until the bottom of the heap $\mathcal{H}$ is reached. Hence,

Theorem 2.8. *There is a parallel algorithm for deleting the smallest element in a heap of size n, running in $\mathcal{O}(1/\varepsilon)$ time with n^ε processors and $\mathcal{O}(n^\varepsilon/\varepsilon \log n)$ extra bits, for any constant $0 < \varepsilon \leqslant 1$, on the CRCW-PRAM model.*

With this approach, we can design a parallel deletion algorithm with an even better time-processor product. In fact, the problem of deleting the smallest element in a heap can be solved in sublogarithmic time by using a sublogarithmic number of processors. Precisely, if we employ $\lfloor \log n/\log\log n \rfloor$ processors and let k (in Algorithm 2.6) be the left match of $\lfloor \log n/\log\log n \rfloor$, then Algorithm 2.6 runs in $\mathcal{O}(\log n/\log\log n)$ steps. That is,

Theorem 2.9. *There is a CRCW-PRAM algorithm for deleting the smallest element in a heap of size n that runs in time $\mathcal{O}(\log n/\log\log n)$ on $\log n/\log\log n$ processors and with $\log n/\log\log n$ extra bits.*

Similarly, we can implement the algorithms developed above on the CREW-PRAM model and achieve the same time-processor products. Namely, we have

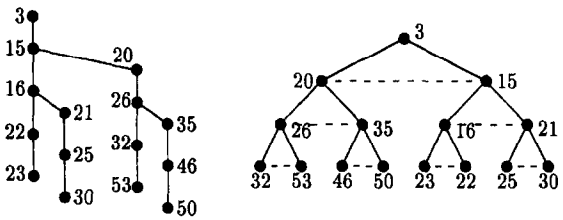
Theorem 2.10. *There is a CREW-PRAM algorithm for deleting the smallest element in a heap of size n that runs either in time either*

- *$\mathcal{O}(1/\varepsilon)$ with n^ε processors n^ε extra bits, for any constant $0 < \varepsilon \leqslant 1$; or*
- *$\mathcal{O}(\log n/\log\log n)$ on $\log n/\log\log n$ processors using $\mathcal{O}(\log n)$ extra bits.*

Remark that finding the path of minimum children (which may be implicit in some heap-deletion algorithm) seems to be almost sequential in nature. More precisely, the computation of the minimum-children path needs to process the heap level by level, and the result of some operation influences latter operations. Hence, the delete operation does not lead to a good parallel implementation.

3. Fine-heap with application

In this section, we shall introduce a new variant of the conventional heap that allows a quick access of the path of minimum children and admits a moderated heapsort, improving both the time for sorting and the space consumption over the traditional heapsort algorithm and its variants. We first investigate the construction problem for this new structure. Then we show how to implement it so that the desired sorting complexity is achieved.



The Hasse diagram The picture

Fig. 1.

3.1. *Fine-heap and its construction*

A *fine-heap* is a heap with additional ordering relation defined on siblings. See Fig. 1 for an example of fine-heaps.

It costs no comparisons to find the path of minimum children, starting at any node in the structure down to the leaf level in a fine-heap. Such an efficient treatment of the path-finding is accomplished by using $\lfloor n/2 \rfloor$ bits of extra space. The fine-heap has also been introduced implicitly in [10, 13, 19]. The sequential construction complexity of fine-heaps can be estimated using an information-theoretical approach.

Theorem 3.1. *The average (and thus also the worst) number of comparisons necessary to build a fine-heap on n elements is at least* $1.864436\ldots n$ *(ignoring lower-order terms).*

Proof. Let $\mathscr{P}_n$ be a fine-heap on n elements and $\mathscr{P}'_{n-1}$ an ordered structure obtained from $\mathscr{P}_n$ by deleting the root of $\mathscr{P}_n$. Denote by $\ell(\mathscr{P}_n)$ and $\ell(\mathscr{P}'_{n-1})$ the number of permutations of the input elements consistent with $\mathscr{P}_n$ and $\mathscr{P}'_{n-1}$, respectively. Clearly, $\ell(\mathscr{P}_n) = \ell(\mathscr{P}'_{n-1})$. In establishing a lower limit on the construction complexity for fine-heaps, let us check the case when $n = 2^h - 1$ for $h > 0$. Notice that a fine-heap on n elements can be viewed as the first two smallest elements connecting to the structure $\mathscr{P}'_{(n-3)/2}$ and a fine-heap $\mathscr{P}_{(n-1)/2}$. Hence,

$$\begin{aligned}
\ell(\mathscr{P}_n) &= 1 \cdot 1 \cdot \binom{n-2}{(n-3)/2} \cdot \ell\left(\mathscr{P}'_{(n-3)/2}\right) \cdot \ell\left(\mathscr{P}_{(n-1)/2}\right) \\
&= \binom{n-2}{(n-3)/2} \cdot \ell\left(\mathscr{P}_{(n-3)/2+1}\right) \cdot \ell\left(\mathscr{P}_{(n-1)/2}\right) \\
&= \frac{1}{2}\binom{n-1}{(n-1)/2} \cdot \ell\left(\mathscr{P}_{(n-1)/2}\right) \cdot \ell\left(\mathscr{P}_{(n-1)/2}\right) \\
&= \frac{n!}{2n((n-1)/2)!\,((n-1)/2)!} \left(\ell\left(\mathscr{P}_{(n-1)/2}\right)\right)^2
\end{aligned}$$

That is,

$$\frac{\ell(\mathscr{P}_n)}{n!} = \frac{1}{2n}\left(\frac{\ell\left(\mathscr{P}_{(n-1)/2}\right)}{((n-1)/2)!}\right)^2$$

By the information-theoretic lower bound, we know that the minimum number of comparisons, on the average, needed to build a fine-heap $\mathscr{P}_n$ on n elements is at least

$$\log \frac{n!}{\ell(\mathscr{P}_n)} = \log(2n) + 2 \cdot \log \frac{((n-1)/2)!}{\ell\left(\mathscr{P}_{(n-1)/2}\right)} \geqslant n \cdot \sum_{i=2}^{\log(n+1)} \frac{1}{2^i} \log\left(2 \cdot (2^i - 1)\right)$$

which gives $1.864436\ldots n - o(n)$. □

A natural way to build fine-heaps is to construct the structure in a bottom-up fashion (namely, by recursively merging the small parts of the structure), similar to Floyd's heap construction algorithm and its variants [2, 8]. When the algorithm is up to merge two full fine-heaps of height $h-1$ with one singleton element (the cost is denoted by $\mathbb{M}(h)$), the information about the path of minimum children can be deduced from the ordering relations in the structure at no extra cost. For inserting the singleton element into the path and re-establishing the ordering of the structure, $h+2$ comparisons are sufficient to complete the tasks in the worst case. Thus, $\mathbb{M}(i) = i + 2$ comparisons. Therefore, the worst-case number, $\mathbb{F}(h)$, of comparisons for constructing a fine-heap on $n = 2^{h+1} - 1$ elements is at most

$$\mathbb{F}(h) \leqslant 2^h \cdot \sum_{i=1}^{h} \frac{\mathbb{M}(i)}{2^i} = 2^h \cdot \sum_{i=1}^{h} \frac{i+2}{2^i} = 2n - h - 2. \tag{1}$$

Hence, constructing a fine-heap of arbitrary size n will cost at most $2n + \mathcal{O}(\log^2 n)$ comparisons (similar to that for the traditional heap [9]).

Lemma 3.2. *A fine-heap on n elements can be constructed in* $2n + \mathcal{O}(\log^2 n)$ *comparisons in the worst case.*

The complexity of the preceding algorithm exceeds the lower bound (Theorem 3.1) by a constant factor. In order to decrease this factor, we shall design an efficient method for constructing small fine-heaps and use them as basic building blocks for arbitrary large sized fine-heaps. Before proceeding with a presentation of our algorithm, we shall demonstrate that it is possible to construct our building blocks faster than the preceding algorithm does. Observe first that constructing a fine-heap on 7 elements costs 10 comparisons according to Eq. (1), which is only one comparison more than the information-theoretic lower bound. However, by carefully examining the symmetric property of the structure, we can actually create three fine-heaps each of size 7 in only 28 comparisons. The idea behind our fast way of building smaller fine-heaps is gained from the study of the mass production of partial orders. A fine-heap on 7 elements $\{20, 26, 35, 32, 53, 46, 50\}$ and its Hasse diagram are shown in Fig. 1.

Lemma 3.3. *Three fine-heaps each of size* 7 *can be constructed in at most* 28 *comparisons in the worst case.*

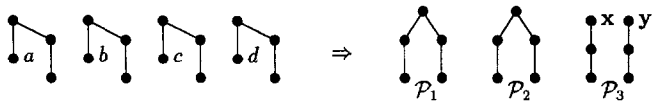


Fig. 2.

Proof. We briefly describe the construction method with the help of Fig. 2.

The algorithm is carried out in four steps.

1. Construct four binomial trees each of size 4. (Cost: 12 comparisons)
2. Compare element **a** with **b**, and element **c** with **d**. (Cost: 2 comparisons) Now the algorithm generates at least the structures $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ (see Fig. 2) plus five singleton elements.
3. Build two 7-element fine-heaps starting from $\mathcal{P}_1$ and $\mathcal{P}_2$ plus two singleton elements for each of the structures. The cost of this step is 10 comparisons, since each singleton element can be inserted into the structure in 2 comparisons (which is carried out by performing a binary search) and $1+1$ more comparisons are needed to achieve the ordering of the fine-heaps.
4. Transform $\mathcal{P}_3$ plus one singleton element into a 7-element fine-heap. To accomplish this step, first a comparison between **x** and **y** is done. Then the singleton element is inserted into the structure (with 2 comparisons) and the ordering property of the fine-heap can then be created with one additional comparison.

The correctness of the algorithm follows directly from its description, and the overall cost is $12+2+10+4=28$ comparisons. □

The complexity of the above algorithm is almost tight, since the information-theoretic lower bound for constructing three 7-element fine-heaps is equal to $\lceil 9+3\times\log 63\rceil = 27$ comparisons. With the similar technique, two fine-heaps of size 7 can be built in at most 19 comparisons, which is also close to the information-theoretic lower bound of $\lceil 6+2\times\log 63\rceil = 18$ comparisons. Although it is not clear how to save more comparisons through producing more fine-heaps each of size 7 simultaneously, our strategy for building 7-element fine-heaps can be used as the building blocks to construct fine-heaps of arbitrary sizes faster than $2n$. In fact, to construct a fine-heap on $n=2^{h+1}-1$ elements, we can apply Eq. (1) and Lemma 3.3, which yields

$$\mathbb{F}(h)\leqslant 2^{h-2}\cdot\tfrac{28}{3}+2^h\cdot\sum_{i=3}^{h}\tfrac{i+2}{2^i}=\tfrac{23}{12}n-h-2.$$

Therefore,

Theorem 3.4. *A fine-heap on n elements can be constructed in at most* $(23/12)n+\mathcal{O}(\log^2 n)$ *comparisons in the worst case.*

This is only 2.8 percent off from the information-theoretic lower bound. The above modified algorithm leads to a slightly better worst-case upper bound on the sorting complexity; as will be shown in the next subsection.

The insert and delete operations on a fine-heap can be performed in a way similar to that for the traditional heap. The parallel complexity of fine-heap operations can easily be deducted from the corresponding time bound for heaps. Moreover, when deleting the smallest element from a fine-heap, one does not need to compute the path of minimum children. Hence,

Observation 3.5. *On the same model of parallel computation,*

- *A fine-heap can be built in parallel as fast as the parallel heap construction using the same number of processors.*
- *The parallel complexity of the insert and delete operations in a fine-heap of size n is the same as that of searching in a sorted list of length* $\mathcal{O}\lfloor\log(n+1)\rfloor$.

3.2. Sorting complexity of fine-heap

Wegener [19] presented a variant of Floyd's heapsort algorithm, which sorts n elements in at most $n\log n + 1.1n$ comparisons in the worst case, using n extra bits and $\mathcal{O}(n\log n)$ bit comparisons. Moreover, it only takes $n\log n + n$ comparisons when $n = 2^h - 1$. Wegener's heapsort algorithm can be viewed as follows:

- Create a fine-heap on n elements in $2n$ comparisons in the worst case, using n extra bits.
- Remove the smallest element from the fine-heap, and repeat this step until there is no element left in the fine-heap.

While a fine-heap of size n can be built in $(23/12)n$ comparisons by Theorem 3.4 (saving $(1/12)n$ comparisons comparing to Wegener's heapsort algorithm), we can also save the amount of extra space consumed using one of the following methods:

1. Implement a fine-heap of size n as a heap with each internal node having an extra bit. This extra bit is used for indicating the smaller child of the node.
2. Implement a fine-heap of size n as a heap, associate a word of length $\lfloor\log n\rfloor - \lfloor\log i\rfloor$ bits to each internal node $\mathscr{H}[i]$. This word will keep the address of the last node on the path of minimum children in the subheap rooted at $\mathscr{H}[i]$.

The first implementation needs $\lfloor n/2\rfloor$ extra bits. However, during the repeated root removals, $\mathcal{O}(n\log n)$ bit comparisons are required in order to follow the path of minimum children. On the other hand, the second implementation takes

$$1\cdot\frac{n}{4} + 2\cdot\frac{n}{8} + 3\cdot\frac{n}{16} + \cdots \doteq n$$

extra bits. With this implementation, no address computation is needed during the sorting phase of the heapsort algorithm. Therefore,

Theorem 3.6. *With our implementations of a fine-heap, Wegener's heapsort algorithm sorts n elements in at most* $n\log n + 1.00274n$ *(and* $n\log n + 0.91667n$ *for* $n = 2^h - 1$*) comparisons in the worst case, using either*

- $n/2$ *extra bits and* $\mathcal{O}(n\log n)$ *2-bit variable comparisons; or*
- n *extra bits and no bit comparison.*

Recently, Dutton [7] presented an interesting sorting algorithm, called weak-heapsort, that makes $n \log n + 0.086n$ comparisons in the worst case. The weak-heapsort uses n additional bits and requires special instructions – boolean functions. The weak-heap introduced in [7] is neither a heap nor a balanced binary tree while the fine-heap is a heap. Unlike the weak-heap, the sequential and parallel implementations of the priority queue operations on the fine-heap can easily be deduced from that for the traditional heap.

4. Conclusions

The goals of this paper are twofold, we provide parallel solutions to the problem of inserting an element into a heap and of deleting the smallest element from the heap, and we introduce a new heap-like data structure that can be used for developing fast sorting algorithm in the fashion of heapsort. The complexity of parallel heap operations and heapsort are improved using extra bits. Interestingly, it is the technique, which builds many isomorphic copies of fine-heaps simultaneously, that leads to a better understanding of their construction complexities. Such a technique needs to be investigated further.

Acknowledgements

The authors would like to thank the referee for helpful comments.

References

[1] S.G. Akl, *The Design and Analysis of Parallel Algorithms* (Prentice-Hall, Englewood Cliffs, NJ, 1989).
[2] S. Carlsson, Average-case results on heapsort, *BIT* **27** (1987) 2–17.
[3] S. Carlsson, A variant of heapsort with almost optimal number of comparisons, *Information Process. Lett.* **24** (1987) 247–250.
[4] S. Carlsson and J. Chen, Parallel constructions of heaps and min-max heaps, *Parallel Processing Letters*, **2** (1992) 311–320.
[5] E.-E. Doberkat, Inserting a new element into a heap, *BIT* **21** (1981) 255–269.
[6] E.-E. Doberkat, Deleting the root of a heap, *Acta Informatica* **17** (1982) 245–265.
[7] R.D. Dutton, Weak-heap sort, *BIT* **33** (1993) 372–381.
[8] R.W. Floyd, Algorithm 245 – Treesort 3, *Comm. ACM* **7** (1964) 701.
[9] G.H. Gonnet and J.I. Munro, Heaps on heaps, *SIAM J. Comput.* **15** (1986) 964–971.
[10] S. Haldar, Heapsort with $n\log(n+1)+n-2\log(n+1)-2$ key comparisons using $\lfloor n/2 \rfloor$ additional bits, Tech. Report RUU-CS-93-14, Department of Computer Science, Utrecht University, Utrecht, The Netherlands, 1993.
[11] D.E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching* (Academic Press, Reading, MA, 1973).
[12] C.P. Kruskal, Searching, merging, and sorting in parallel computation, *IEEE Trans. on Comput.* **C-32** (1983) 942–946.
[13] C.J.H. McDiarmid and B.A. Reed, Building heaps fast, *J. Algorithms*, **10** (1989) 352–365.

[14] M.C. Pinotti and G. Pucci, Parallel algorithms for priority queue operations, in: *Proc. 3rd Scandinavian Workshop on Algorithm Theory*. Lecture Notes in Computer Science, Vol. 621 (Springer, Berlin, 1992) 130–139.

[15] N.S.V. Rao and W. Zhang, Building heaps in parallel, *Information Processing Letters* **37** (1991) 355–358.

[16] R. Schaffer and R. Sedgewick, The analysis of heapsort, *J. Algorithms* **15** (1993) 76–100.

[17] A. Schönhage, M. Paterson and N. Pippenger, Finding the median, *J. Comput. System Sci.* **13** (1976) 184–199.

[18] M. Snir, On parallel searching, *SIAM J. Comput.* **15** (1985) 688–708.

[19] I. Wegener, The worst case complexity of McDiarmid and Reed's variant of BOTTOM-UP HEAPSORT is less than $n \log n + 1.1n$, *Inform. and Comput.* **97** (1992) 86–96.

[20] I. Wegener, BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small), *Theoret. Comput. Sci.* **118** (1993) 81–98.

[21] J.W.J. Williams, Algorithm 232: Heapsort, *Comm. ACM* **7** (1964) 347–348.

[22] W. Zhang and R.E. Korf, Parallel heap operations on an EREW PRAM, *J. Parallel Distributed Comput.* **20** (1994) 248–255.